AMPL Tutorial

Tulia Herrera, IEOR PhD Student

December 25, 2009

1 Introduction

The objective of this tutorial is to provide some basic guidelines on using AMPL. We assume that the student is taking or has taken an Introduction to Mathematical Programming course (e.g. IEOR3608) and also has basic knowledge of any programming language (C/C++, Matlab etc).

This tutorial is divided into three main parts. The first part concerns access to AMPL: where to find it? How to install the student edition? etc. The second part shows examples in a very basic way; how to translate our mathematical programming problems into AMPL programming language. The last part focuses on *scripts* and useful commands to integrate your optimization problem with other elements of a programming environment.

2 Availability of the Software

AMPL is available at the IEOR computer lab. If you would like to program using your own computer there is an **AMPL student edition** that can be downloaded for free. As you might expect the student edition has some restrictions related with the number of variables and constraints (up to 300 constraints and 300 variables) of your problem.

As a final note, AMPL is not a solver. It is a modeling language for mathematical programming. This means that you program in AMPL and then AMPL translate your code to a solver like CPLEX or MINOS.

2.1 AMPL In the Laboratory

After logging into your laboratory account, you should go to **start**, then **all programs** and then you will find **ILOG**. **ILOG** is an authorized reseller of AMPL, which was developed at Bell Laboratories. **ILOG** will display some options; you should click in the **AMPL 11.2** option which will display some other options, then select **ampl.exe ver 11.2**. After choosing this option, AMPL will be running!!.

A note on the location of your files. Since you will be working in a public location, you have two options. One is to run your models from your USB memory or the location of your preference, in which case you will require to type the access route of your file when you load your model. The other option is to temporary copy your models in the **content folder**. This is one of the options that was displayed when you clicked **AMPL 11.2** option. In this case, you can refer to your files directly by their names without specifying the location.

2.2 AMPL student edition

2.2.1 Downloading and Starting AMPL student edition

The AMPL student edition is available to download from AMPL website:

`http://www.ampl.com/DOWNLOADS/index.html`

Here, you will be able to download **amplcml.zip** file, which you should unzip in the location of your preference. This folder contains AMPL and some solvers as MINOS 5.5 (minos.exe), CPLEX 10.1 (cplex.exe and cplex101.dll), and the Kestrel client (kestrel.exe) which allow access to other solvers via the Internet.
To start AMPL you should open the file **sw.exe**, located in the **amplcml** folder (name by default) that you just unzipped. Then you should type:

```
ampl
```

Now, AMPL is running!

3 Mathematical Programming Problems in AMPL

Consider the following problem and its corresponding LP formulation:

Columbia Candies Factory produces two flavors of chewing gum: mint and cinnamon. The mint flavor gum is sold for $1 per package, while the cinnamon gum is sold for $1.5 per package. The company has limited resources and can produce only one flavor of gum at a time. The rates of production are different; the company can produce 40 package of mint gum per hour but only 30 packages of cinnamon gum per hour. Considering the result of a marketing study, the company decided that at most it can sell 900 packages of cinnamon gum and 1000 packages of mint gum. If we consider that a week has 40 hours of labor and we have no storage capacity, we need to determine how many packages of mint and cinnamon gum we should produce so that the total revenue is maximized.

The corresponding LP formulation is shown below:

$$\begin{array}{rl} \textit{maximize} & x_m + x_c \\ \text{s.t.} & \frac{1}{400} \cdot x_m + \frac{1}{300} \cdot x_c \leq 40 \\ & x_c \leq 900 \\ & x_m \leq 1000 \\ & x_m \geq 0 \\ & x_c \geq 0 \end{array}$$

3.1 Writing AMPL code

To write your code you can use any text editor (e.g. Notepad) but given that you are programming, we recommend you to use a source code editor (e.g. crimson editor). One of the advantages of source code editors is the use of **syntax highlighting**, which helps you catch syntax errors. There are a considerable number of free source code editors available in the Internet.

It is important that you know that the use of a text editor is not mandatory; it is just a recommendation to organize and make easier your programming tasks. In fact, you can type the code directly in the **AMPL command window** – in that case you may not be able to save your work. However, it is a very useful tool for debugging tasks.

3.2 from LP formulation to AMPL code

3.2.1 Direct formulation

The natural approach to program this LP formulation in AMPL is entered using syntax that is very close to the algebraic expressions of the LP formulation above. To start you need to create a **.mod** file, for example *example1.mod*, in which you should write the AMPL code. In general you can divide the code in three main parts as shown below:

```
# PART 1 DECLARATION OF VARIABLES (variables, parameters, sets etc)
var x_c >= 0;
var x_m >= 0;
# PART 2 OBJECTIVE FUNCTION: name and mathematical expression
maximize revenue:  x_m + 1.5*x_c;
# PART 3 CONSTRAINTS: names and corresponding mathematical expressions
subject to Aval_Time:  (1/40)*x_m+(1/30)*x_c<=40;
subject to Max_Mint:  x_m<=1000;
subject to Max_Cinn:  x_c<=900;
```

Comments

- The symbol `#` indicates the start of a comment. The remainder of the line, after the `#`, is ignored.
- Variables must be declared using the `var` command.
- All lines of code must end with a semi-colon (`;`).
- The objective function starts with the command `maximize` or `minimize`, followed by the name of the objective function (cost, revenue, etc), followed by a colon (`:`) which is finally followed by the function that should be optimized, terminated by the corresponding semicolon (;).
- Each constraint or array of constraints (constraints under the same name and different index) starts with the command `subject to` followed by a name, followed by a colon (`:`) and finally followed by the corresponding logical constraint.
- Names are unique. Variables, constraints and the objective function muat have different names.
- AMPL is case sensitive. Commands must be in lower case. This also applies to name of entities, i.e `a` and `A` are different names.

3.3 Solving the problem

3.3.1 Approach 1: Without `scripts`

If you saved your **.mod** file in the folder **amplcml** (which contains the **sw.exe** file and the solvers), you will not need to refer to your **.mod** file using its path. Assuming that this is your case, the first step in solving your problem is to load the model: variables, objective function and constraints. To load your model you should type in your **AMPL command window**:

```
model example1.mod;
```

On the other hand if your **.mod** file is in a different location you should refer to it using its actual path. For example, if you saved your **.mod** file in `Users` then you should use the following:

```
model C:\Users\example1.mod;
```

If there is a syntactical mistake in your model, AMPL will display an error message. Debugging errors in AMPL is not always an easy task, as the error messages do not necessarily provide a lot of information.

After loading your model, you should type `solve;` to solve your problem. Once the problem is solved you can ask for variable values. In the above example we wanted to find out the number of packages of mint gum and cinnamon gum that you should produce; you could use the command shown below to display these results:

```
display x_c, x_m;
```

3.3.2 Approach 2: Using `Scripts` (.run file)

Sometimes we need to repeatedly solve the same problem (perhaps with slight modifications). Typing lengthy sequences of commands in the **AMPL command window** can become tedious. You can avoid this by using a **script**. Scripts capture in a file a sequence of commands that can be repeatedly used by calling the **script** function in the **AMPL command window**.

Returning to the example, to make use of a **script** you should open a new file; assume you name it *example1.run*. In this file you are going to write the sequence of commands that you want to perfom. For our example you will write:

```
#RESET
reset;
#LOAD THE MODEL
model example1.mod;
#CHANGE THE SOLVER (optional)
option solver cplex;
#SOLVE
solve;
#SHOW RESULTS
display x_c,x_m;
#OTHER OPTIONS TO DISPLAY
display revenue, Aval_Time, Max_Mint, Max_Cinn;
show revenue, Aval_Time, Max_Mint, Max_Cinn;
```

After you save your **.run** file, you just need to refer to it in the AMPL command window in order to perform all the tasks that you asked for in the **.run** file.

```
include example1.run;
```

Comments

- Since you are interested in loading your model more than once (changing coefficients, after correcting some errors, etc.), you will need to periodically `reset` AMPL. You cannot load your model with variables, constraints, or objective function using names which already exist – “reset” allows AMPL to forget all names.
- By default the **AMPL student edition** uses, as solver, MINOS 5.5, but if you want to use (e.g.) CPLEX as solver, you should define CPLEX as your solver (using `option solver cplex;`). You will just have to perfom this task once every time that you open the **AMPL command window**.

- In general you could use the command `display` followed by the name of the entity whose value you are interested in: objective function, variables, dual variables etc.
- To display the algebraic expression of your objective function or constraints you shoud use the command `show` followed by the name of the objective function, or the corresponding constraint. The command `expand` have a similar use.
- To execute the actions indicated in the **.run** file you have to type the `include` command before the name of the **.run** file. The include command does not require a semicolon at the end.

3.4 Taking advantage of AMPL

Writing a mathematical programming problem with a large number of constraints (e.g. 100, 1000 ...) would be very taxing if you had to type each constraint as in the previous example. For example, suppose that instead of having just two products, Columbia Candies Factory had a very wide line of candies, perhaps 10, 100 or even larger. AMPL lets you formulate the problem in a more general form.

Given:
n = number of flavors
t = total labor hours in a week.
p_i = profit per package of gum with flavor i, i = 1, . . . , n.
r_i = Number of packages of gum flavor i produced per hour, i = 1, . . . , n.
m_i = maximum demand for flavor i, i = 1, . . . , n.
Variables: x_i = Number of package of gum flavor i to be produced, i = 1, . . . , n.

$$\begin{array}{rl} \textit{maximize} & \sum_{i=1}^{n} p_i \cdot x_i \\ \text{s.t.} & \sum_{i=1}^{n} r_i \cdot x_i \leq t \\ & 0 \leq x_i \leq m_i \text{ for each } i = 1..n. \end{array}$$

In order to take advantage of this general form, we will write the model using notation that avoids explicitly writing the constraint for each different flavor. Here you are going to use two different files, a **.mod** file and a **.dat** file.

In the **.mod** file, you describe the problem in a fairly generic form, describing the objective function and constraints in such a way that you can change the number of flavors, revenue per flavor and any other data without changing this **.mod** file. To achieve this goal, you create a new file named *example2.mod*, whose contents will be:

```
#Declaration of parameters
param n;
param t;
param p{i in 1..n};
param r{i in 1..n};
param m{i in 1..n};
#Declaration of variables
var x {i in 1..n} >=0;
#Objective Function
maximize revenues:  sum {i in 1..n} p[i]*x[i] ;
#Constraints
subject to Aval_Time:  sum{i in 1..n} (1/r[i])*x[i];
subject to Max_Flavor {i in 1..n}:  x[i]<=m[i];
```

After doing this you should verify that you can load your model:

```
reset;
model example2.mod;
```

If AMPL reports some errors you should correct them to guarantee that your model can be loaded. The model can still have modeling problems, but at least at this moment AMPL is able to read what you wrote.

After verifying that your model is correct you should create a **.dat** file. This file will contain all data values arising in your model. An example of **.dat** file, corresponding to *example2.mod*, is shown next:

```
param n := 4;# No of Flavors
param t := 40; # Total labor hour in a week
param p := 1 1 2 1.5 3 1 4 1.5; # Revenue per package flavor i
param r := 1 40 2 30 3 50 4 20; # Production rate of package flavor i
param m := 1 1000 2 900 3 500 4 800; # Maximum demand package flavor i
```

Assuming this file has been named *example2.dat*, we can load it and after that, solve the problem:

```
data example2.dat;
solve;
```

You can also creatre a **.run** file and include in it the above commands: load the model, load the data, solve the problem, etc. The (*example2.run*) file would be something like:

```
reset;
model example2.mod;
data example2.dat;
solve;
display x;
```

Comments

- Given that AMPL ignores carriage returns (in files) and instead looks for semicolons to terminate lines, you could also write the one-dimensional array paratemers as:

```
param p :=
1 1
2 1.5
3 1
4 1.5;
```

- When a parameter has more than one element (but is still a one dimensional array) each value is preceded by the corresponding index.
- When you have the same constraint for some elements of a set, you can group the constraints under a common name and index each element of the set. This is called an *array of constraints*.
- Given that x is an array of variables, in order to display the values of all x variables you should just type: `display x;`. If you would like to display a specific variable, for example x[2], you should type: `display x[2];`.

In general, in the **.dat** file you assign values to the parameters and constant components that you previously declared in the **.mod** file. AMPL looks in the **.dat** file for the parameters or constant components that you declared in the **.mod** file to assign the corresponding values when you request to solve the problem.

An alternative version of the **.dat** file can be created using a single matrix parameter, instead of the three array parameters (p,r,m). This matrix will contain all the information per flavor (revenue, rate and maximum demand). The matrix will have n rows and 3 columns. Each column corresponds to one of the parameters (p, r, m).

In this version we also introduce a new component: `set`, it is used to group a finite number of elements under a common name. In our example, we first declare the `set N` (in the **.mod** file), and then after we assign as elements the integers from 1 to n (in the **.dat** file). We also declare the `set INF` (in the **.mod** file), and then we assign as elements the labels of the three characteristics (p, r, m) of the flavors (in the **.dat** file). For more details about `set`, see Section 4.1.1.

Finally, to operate in this manner, we need to alter the model file. Let us create a new file, *example3.mod* (the corresponding data file is given later):

```
param n;
param t;
set N := {1..n};
set INF;
param information{N,INF};
#Declaration of variables
var x {i in N} >=0;
#Objective Function
maximize revenues:  sum {i in N} information[i,'p']*x[i] ;
#Constraints
subject to Aval_Time:  sum{i in N} (1/information[i,'r'])*x[i]<=t;
subject to Max_Flavor {i in 1..n}:  x[i]<=information[i,'m'];
```

Comments

- Use **sets** to group some elements; the elements can be indexes (as in our example), name of products, etc.
- To index elements of a set, for example in a summatory, the elements are not necessarily numerical.

In the corresponding data file, **example3.dat**, is used to introduce (in addition to the old parameters) the new parameter “information”, which is a $n \times 3$ matrix, indexed by the elements of two sets, `N` and `INF`, one with numerical and the other with non-numerical elements.

```
param n := 4; #No of Flavors
param t := 40; #Total labor hour in a week
set N := 1..n;
set INF := p r m;
#Information matrix:  profit, rate and maximum demand per flavor
param information:
 p r m :=
1 1 40 1000
2 1.5 30 900
3 1 50 500
4 1.5 20 800
;
```

Comments

- As we declare in the **.mod** file, the rows of the `param information` correspond to the elements of the set N and the columns correspond to the elements of the set INF.
- The first row of the information parameter correspond to the labels assigned to the three characteristic of the flavors (p,r,m). These labels correspond to the elements of the set INF.
- The first column of the information parameter correspond to the index assigned to each flavor. These indexes correspond to the elements of the set N.
- Its very important to take care of the syntax that AMPL require to introduce a matrix, there is a `:=` symbol following the labels of the columns and then there is just one a semicolon (;) at the end of the matrix.

As before, to solve the problem, you can simply create an appropriate **.run** file.

We now have a basic view of how AMPL works. The remainder of the tutorial focuses in some examples of common and useful commands in AMPL.

4 Some Useful Commands in AMPL

4.1 Components

4.1.1 Sets

AMPL *sets* can be used to define finite collections of elements. These elements can be numerical or non-numerical, and frequently are used as indices. Table set shows some examples of set declarations (in the **.mod** file), and the assignment of value (in the **.dat** file).

Table 1: Examples of sets in AMPL.

.mod FILE	**.dat FILE**

`param n;`
`set P := {1..n};`
	`param n:= 4;`

`set Q;`	`set Q:= 1 2 3 4;`
`set R;`	`set R := rainy, cloudy, sunny;`

`param t;`
`set T := {1..t};`
`set U{T} default {};`
	`param t := 1000;`

In the last example, we used the statement `default` follow by {}. This means, that after defining the set `U`, whose elements are indexed by the set `T`, we state that `U` starts empty.

4.1.2 Parameters

The *parameter* entity is used to hold constant values. In our example, parameters are used for the total number of gum flavors, the profit per package, the rates of production and the maximum produce amount per flavor. Table 2 shows some examples of *parameter* declarations (**.mod** file) and the corresponding assignment of values (**.run** file).

Table 2: Examples of parameters in AMPL.

.mod FILE	**.dat FILE**
`param m1;`	`param m1 := 1 1.2 2 1.5 3 1.4;`
`param m2;`	`param m2 :=`
	`1 1.2`
	`2 1.5`
	`3 1.4;`
`set M;`	`set M:= a b c d;`
`set O;`	`set O:= e f g h;`
`param p{M,O};`	`param p:`
	`a b c d :=`
	`e 1 5 3 2`
	`f 4 3 1 0`
	`g 3 2 7 8;`
`param n;`	`param n:= 3;`
`set M;`	`set M:= a b c d;`
`set O;`	`set O:= e f g h;`
`set N;`	`set N:= {1..n};`
`param q{M,O,N};`	`param q:`
	`[*,*,1]:  a b c d :=`
	`e 30 10 8 10`
	`f 22 7 10 7`
	`g 19 11 12 10`
	`[*,*,2]:  a b c d :=`
	`e 39 14 11 14`
	`f 27 9 12 9`
	`g 24 14 17 13`
	`[*,*,3]:  a b c d :=`
	`e 41 15 12 16`
	`f 29 9 13 9`
	`g 26 14 17 13;`

Comments

- In general, parameters can be indexed with cartesian product of sets.

4.1.3 Variables

Variables are the AMPL entities representing the actual variables in our mathematical program. We illustrate the declaration of variables in a **.mod** file (you can also do this in a **.run** file) through examples. See Table 3.

Comments

- Bound constraints over the variables can be imposed in the declaration of the variables in the **.mod** file. In the third example in Table 3 we imposed upper and lower bounds over the vector of variables `x` during the declaration of the variables.
- By default the variables are assumed to be real numbers, if your problem requires that a variable or vector of variables is restricted to be `integer` or `binary` you should write the command `integer` or `binary` in front the declaration of the variable.

Table 3: Examples of variable declarations in AMPL.

`var x;`
`var x{N};`
`var x{1..n} >=0 <=1;`
`var x{1..n,M} integer;`
`var x{M,O,N} binary;`

- In general, variables can be indexed with cartesian product of sets.

4.1.4 Constraints

Here we provide examples of how to express constraints. Recall that in our gum problem we have `n` different gum flavors; for each we have to define a constraint to handle maximum demand. Writing each constraint "by hand" is a very tedious task, in fact very much so when `n` is large. Using some examples we will next how how can you use AMPL to setup arrays of constraints, etc. See Table 4.

Table 4: Examples of constraint arrays in AMPL.

.Mod FILE
`subject to constA {i in 1..n}:  x[i]<=b[i];`
`subject to constB {i in N}:  x[i,'a']+x[i,'b'] <=c[i];`
`subject to add_ConstC {l in MAXCONST : card(CONST4[l]) <> 0}: I4[l,1]*bx[1]+I4[l,2]*(1-bys[1])<= I4[l,3]+I4[l,4]*ly[1];`

The last example in Table 4 represents an array of constraints which size (number of constraints under the name `add_ConstC`) is given by the size of the set CONST4, however the maximum size that the array `add_ConstC` can achieve is given by the number of elements of the set MAXCONST. This statement create a constraint only if the element `l` in `MAXCONST` set have an image in the `CONST4` set (we can assume that the relation in 1-1). In other words, if for the element `l`, there exist one element CONST4[l] then we create a constraint that correspond to the element `l`. If by the contrary, for `l` in `MAXCONST`, `CONST4[l]` is empty, then we don't have a constraint for `l`. As additional note, if you want to declare a set to be empty at start you can use the command `default` in front the declaration of the set and before the semicolon (;).

This kind of statements are very useful when you want to add new constraints to a problem in an automatic way depending on some previous results. For example in an integer programming problem you want to add some valid inequalities to cut your feasible region given than the solution of our LP relaxation is not integer.

4.2 Scripts

As mentioned above, a script is a sequence of commands captured in a file. Scripts are very useful because they can contain many programming language features such as loops, if-else statements, other scripts and many AMPL commands. We ilustrate some of these structures using examples which focus on the syntax.

In the context of *scripts* it is important to keep in mind that parameters, sets, variables, constraints and objective functions are global entities. So you can not re-declare a variable or parameter that has already been declared.

In fact, it can be a common problem when working with AMPL that onee tries to declare a variable that already has been declared. To avoid this problem, we recommend that all declarations be done in a *script* file and not in the **.mod** file.

4.2.1 The Include and Commands statements

These two statements function as *scripts*. Using programming language notation, they are similar to "functions" that you call to execute a sequence of commands. An example in which we use the command `include` is with the **.run** file, as we saw above.

```
include example1.run;
```

`commands` is a statement very similar to `include`. However this is a true statement that needs a terminating semicolor (;) and can only appear in a context where the statement is legal, for more information see [1]. To create a `commands` file you should create a file (with any extension). Then, you can write a sequence of commands that you would like to execute when you call your script:

```
commands name.flag;
```

In general `commands` scripts are used in the main script to execute a group of commands that you will require to execute more than once. Speaking in programming language, `commands` scripts perform the work of a "function".

4.2.2 The Problem statement

This statement is mostly used when you have more than one optimization problem in the same *script*. For example, first you need to solve one problem, and then using some results from that first problem you need to solve a second one. The "problem" command is followed by the problem name, i.e `problem2`, and then is followed by a colon (:). After the colon we state the objective function, constraints and variables (in any order) by the name that they have in the **.mod** file. The example below, *example4.run*, shows a `problem` statement for *example2*, see Section 3.4.

```
reset;
model example2.mod;
data example2.dat;
problem problem2:  revenues, Aval_Time, Max_Flavor, x;
solve problem2;
expand problem2;
display x;
```

Comments

- Suppose you have an array of `n` constraints. To include all of them you just have to type the name of the array of constraints. But if you actually just want to include some of them, you should write the name with the corresponding index for example Max_Flavor[1].

4.2.3 Iterating : FOR and REPEAT statements

These are looping commands to repeatedly perform tasks until some condition holds. The `for` statement executes a statement or collection of statements for each member of some set (remember that such members can be numerical or non-numerical). Here are some examples:

```
for {1..4} {
solve problem1;
let p1[1] := p1[1]+1;
solve problem2;
let p2[1] := p2[1]+1;
}
```

In general, the `for` statement syntax is:

```
for {i in N} {
...
}
```

In the case of the `repeat` statement, the iterations continue as long as some logical condition holds. An example from [1]:

```
repeat while T[2].dual > 0 {..};
repeat until T[2].dual =0 {..};
repeat {..} while T[2].dual >0 ;
repeat {..} until T[2].dual >0;
```

Here, the loop body is inside the {}. All these four statements are different. When the body appears after the `while` or `until` statement, the validation of the condition is made before the a pass through the body commands. When the body appears before the `while` or `until` the validation is performed after an execution of the body commands.

The `while` statement repeats while the validation condition holds; on the contrary the `until` statement repeats while condition is false. In our example, while the dual variable correspoding to the constraint T[2] is greater than zero the loop continues. In the `until` statement, so long as the dual variable corresponding to constraint T[2] is different from zero, the loop execution continues.

As additional information, the command `let` is used to change the value of an entity. For example, if you have a flag and you want to update his value to 1, you just have to type `let flag:= 1;`.

4.2.4 IF-THEN-ELSE statement

This statement is used to describe conditional expressions; its general syntax is as follows:

```
if (logicalcondition1) then {
..
}
else if (logical condition2) then {
..
}
else {
..
}
```

Comments

- The logical condition is any statement that can be true or false. For example $a < 100$, $b[1] < 1$ and $b[2] < 1$, $b[1] < 1$ or $b[2] < 1$.

- Note that there is no semicolon (;) after the brackets that enclose the body. The semicolon just follows commands.
- If the body is a single command you do not need to use brackets.

4.2.5 Some DISPLAY options

Table 5 shows some basic examples of output options, for example how to save your outputs to a file, or how to display them on the screen.

Table 5: Some examples of display options.

Saving in a file	#This will overwrite the file if it already exists.
	display PaintB > z:\myFiles\example1.out;
	#add to an already existing file.
	display PaintB >> z:\myFiles\example1.out;

printf and printf	# to print a 3×4 matrix stored as parameter a[i,j].
	print {i in 1..3}:{j in 1..4} a[i,j];
	#Print a column with the numbers 1 through 3.
	printf {i in 1..3}: \"%3i\n", i;

5 References

[1] Fourer, R., Gay, D. & Kernighan, B. (2002). A Modeling Language for Mathematical Programming. (Second Edition). Duxbury Press, Brooks/Cole Publishing Company.